

# PRELIMINARY REPORT ON EXTRACTING OBJECT-BASED DESIGN FROM FUNCTIONALLY-ORIENTED IMPLEMENTATIONS

**SPC-92088-CMC**

**VERSION 01.00.06**

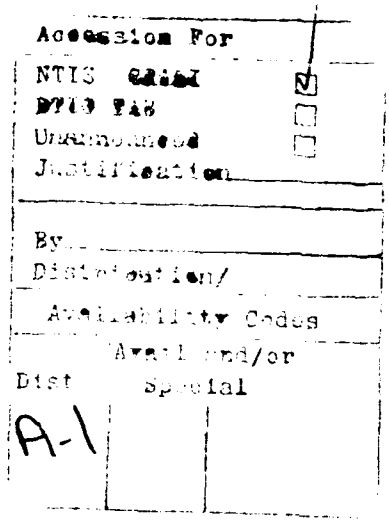
**NOVEMBER 1992**





# PRELIMINARY REPORT ON EXTRACTING OBJECT-BASED DESIGN FROM FUNCTIONALLY-ORIENTED IMPLEMENTATIONS






## SPC-92088-CMC

## VERSION 01.00.06

## NOVEMBER 1992

**Jeff Facemire**

Produced by the
SOFTWARE PRODUCTIVITY CONSORTIUM SERVICES CORPORATION
under contract to the
VIRGINIA CENTER OF EXCELLENCE
FOR SOFTWARE REUSE AND TECHNOLOGY TRANSFER

SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070

CDOC is a trademark of Software Blacksmiths, Inc.

# CONTENTS

# FIGURES

*This page intentionally left blank.*

# ACKNOWLEDGEMENTS

This report documents a method that was developed by Thomas Pole (formerly of the Software Productivity Consortium, but currently with EVB Software, Inc.) in conjunction with the SYSCON pilot project. As such, this report utilized the following as the basis for the report:

- A prototype tool that implements the techniques of this method.
- A draft external paper entitled: "Transitioning to the Object Oriented Software Development Paradigm Using C2C++: Recovering the Implicit Reusable Objects from a Non-Object Oriented Implementation" written by Thomas Pole.
- Conversations between the author, Thomas Pole, and Tom Barr (SYSCON Corporation).

I also wish to thank Tom Barr (SYSCON), Lisa Finneran, Fred Hills, Jim O'Connor, and Thomas Pole for being reviewers of this paper. Their feedback was essential.

*This page intentionally left blank.*

# EXECUTIVE SUMMARY

This report describes preliminary work on a method for extracting *object-based design* from *functionally-oriented* implementations. This method is applied to existing functionally-oriented code to produce a set of objects that are behaviorally-equivalent to the original implementation. Each of these objects represents a *cohesive* grouping of related functions and the data that the functions manipulate.

The purpose of this report is to sufficiently explain this method so that it serves as the basis for future work. This method is preliminary in nature, as it has only been validated on a single project. It was developed and practiced on a pilot project between the Consortium and the SYSCON Corporation. As such, the intent of the report is to help the audience to "understand" the method and to serve as the basis for further validation/exploration. This report is not intended to instruct you on how to apply the method to new projects. The primary audience is technologists and methodologists who are interested in exploring methods for transforming functionally-oriented implementations into reusable objects or object-based implementations.

This report provides a systematic method for extracting an object-based design, implicit within the functionally-oriented implementation, that exhibits the benefits of encapsulation, information hiding, and problem space orientation. This method shows you how to analyze the organization of the existing implementation and to form cohesive objects with well-defined interfaces. The objects identified by this method partition the original functions and data into cohesive, logically-related groups. The resulting design is considered object-based, instead of object-oriented, because it focuses on encapsulation and information hiding as opposed to the other object-oriented characteristics such as inheritance and polymorphism.

*This page intentionally left blank.*

# 1. INTRODUCTION

## 1.1 PURPOSE

This report describes preliminary work on a method for extracting *object-based design* from *functionally-oriented* implementations. This method is applied to existing functionally-oriented code to identify a set of *objects* that are behaviorally-equivalent to the original implementation. Each of these objects represents a *cohesive* grouping of related functions and the data that the functions manipulate.

The purpose of this report is to sufficiently explain this method so that it serves as the basis for future work. This method is preliminary in nature, as it has only been validated on a single project. It was developed and practiced on a pilot project between the Consortium and the SYSCON Corporation. As such, the intent of the report is to help the audience to "understand" the method and to serve as the basis for further validation/exploration. This report is not intended to instruct you on how to apply the method to new projects.

## 1.2 INTENDED AUDIENCE

The intent of this report is to document the method practiced on the SYSCON pilot project and to serve as a basis for further exploration. Therefore, the primary audience is technologists and methodologists who are interested in exploring methods for transforming functionally-oriented implementations into reusable objects or object-based implementations.

However, line engineers may also find this method, in its current form, useful for extracting object-based design information from existing functionally-oriented implementations. The major caveat to line engineers, though, is that the material has only been validated on the SYSCON pilot project. As such, the Consortium cannot predict **a priori** that this method (without some form of customization) will yield the same results as those found by SYSCON. To be fully functional to line engineers, this method still needs to be applied on additional pilot projects to ensure maturity of the method and to better understand any needed customizations.

To be applied effectively, this method requires domain experts to be involved in examining the method's intermediate and final results. An intimate knowledge of the implementation's design and architecture is necessary to accurately assess whether the resulting objects are what you want. This method also requires someone to apply it either manually or through some automated means. Much of this method would benefit from using automation; however, the method can be applied manually.

## 1.3 MOTIVATION AND GOALS BEHIND THE METHOD

Even though the title of this report uses the term "extracting," this method shares much of its motivation and goals with that of *reengineering*: namely, the desire to transform existing code into

better code or to improve its understanding. There are a number of reasons why one might want to reengineer existing systems. Some of these reasons include:

- A requirement to support a system over a long period of time. Reengineering could be used to make the existing code more maintainable.

- Systems that are poorly documented. Reengineering could be used to extract valuable implementation interactions that are not immediately obvious from looking at large amounts of code.

- A desire on the part of a company to develop many similar systems. Reengineering existing code could provide a source of reusable components.

Each of these reasons can benefit from the use of some method for extracting information from existing code that is not currently (or readily) available.

This method focuses on functionally-oriented implementations that are systems where the "boundaries of modules have been defined in a way that depends on the decomposition, which in turn depends on the functional characteristics of the specific application." (Graham 1991) Functionally-oriented implementations are based on an organization of functional modules that are defined as subsets of a implementation's functionality, identifying each module by a set of high-level functional abstractions representing that module's subset of the system functionality. The data, and therefore the state of the system, are not explicitly included in the module (i.e., it is not based on *information hiding*).

The primary problems inherent in functionally-oriented implementations is that maintenance is difficult and reusing modules is inhibited. Maintenance is usually the major portion of the lifecycle for any given system. As such, enhancements and fixes that take place throughout the maintenance lifecycle often introduce more work than they save because the code is hard to understand, implications of change are not clear, and changes are not isolated to a particular part of the system. Reuse of modules is usually inhibited because of the lack of minimalized dependencies between functional modules.

An object-based orientation to system organization alleviates these problems, while also addressing the reasons for performing code reengineering, by offering the following:

- **Encapsulation.** *Encapsulation* is a technique of isolating a system function within a module and providing a precise specification for the module (IEEE 1983). Encapsulation groups related functions and data together so that they can be treated and thought of as a unit. Encapsulation is very closely related to information hiding.

- **Information Hiding.** Information hiding is a technique of encapsulating software design decisions in modules in such a way that the module's interfaces reveal as little as possible about the module's inner workings; thus, each module is a "black box" to the other modules in the system. The discipline of information hiding forbids use of information about a module that is not in the module's interface specification (IEEE 1983).

- **Problem Space Orientation.** Problem space orientation organizes system objects around real-world objects such as external environment entities, hardware components, and user

operations. This orientation allows the user to better understand the relationship between modules of an implementation and the real-world functions addressed by the modules.

Encapsulation, information hiding, and taking a problem space orientation provide the basis for creating software that is more reusable, extensible, and maintainable as follows:

- **Enhanced Reuse Potential.** An object-based orientation groups data and functionality together that cohesively make sense. Proper balance of encapsulation and information hiding will result in objects that are thought of as a unit and contain well-defined interfaces for understanding the exact nature for interacting with the object. A problem space orientation increases the chances of (and the understanding of) how the software can be reused in future systems. This problem space orientation makes it easier to map from future requirements to existing implementations that address the requirements.

- **Increased Extensibility.** An object-based orientation establishes objects that are cohesive, with respect to function and data, with well-defined interfaces for defining the objects. The cohesiveness and the strict interfaces make objects easier to understand which, in turn, makes them easier to change. This ease of understanding is also enhanced by the problem space orientation since the mapping from the problem space to the implementation is easier to make. Proper encapsulation and information hiding mean that changes to particular objects will not adversely affect other objects as long as the changes are confined within the particular object. Only when changes affect an object's interface is it necessary to investigate the changes' impact on the rest of the system.

- **Increased Maintainability.** An object-based orientation increases maintainability by establishing objects with well-defined interfaces for easier localization of change. As with extensibility, the objects are also easier to understand which is vital when trying to maintain a system long after its original development phase. Taking a problem space orientation also makes it easier to map future enhancements to existing objects that are effected.

This report provides a systematic method for extracting an object-based design, implicit within the functionally-oriented implementation, that exhibits the benefits of encapsulation, information hiding, and problem space orientation. This method shows you how to analyze the organization of the existing implementation and to form cohesive objects with well-defined interfaces. The objects identified by this method partition the original functions and data into cohesive, logically-related groups (i.e., functions that tend to manipulate the same data or call the same functions would be good candidates for being placed in the same object). The resulting design is considered object-based, instead of object-oriented, because it focuses on encapsulation and information hiding as opposed to the other object-oriented characteristics such as inheritance and polymorphism.

## 1.4 TYPOGRAPHIC CONVENTIONS

This report uses the following typographic conventions:

| | |
|---|---|
| Serif font ..... | General presentation of information. |
| *Italicized serif* font ..... | Words, expressions, abbreviations, and acronyms found in the Glossary, and publication titles. |

**Boldfaced serif** font ..... Section headings and emphasis.

**Boldfaced sans serif** font ..... Within commands, commands and keywords to be used literally.

`Typewriter` font ..... Syntax of code or software responses.

# 2. THE CONCEPTUAL METHOD

This section describes the method for extracting object-based design information from functionally-oriented implementations. It initially presents an overview of the method that will introduce most of the method-specific terminology used later in the section. The method's individual activities follow the initial overview.

## 2.1 METHOD OVERVIEW

As the title suggests, this method extracts an object-based design from a functionally-oriented implementation. A functionally-oriented implementation is made up of *data elements* (i.e., variables, records) and *functions* (i.e., procedures, functions), collectively referred to as *components*. This method uses interactions between components as the basis for forming an object-based design that is a behaviorally-equivalent representation of the original functionally-oriented implementation.

The object-based design is derived from the functionally-oriented implementation by analyzing the interactions between components to determine if sufficient *commonality* exists to warrant forming an object to contain the components. Commonality in this method is defined as the common calls or data accesses between two components. In other words, if two components primarily call the same functions or access the same data, then they are likely to belong in the same object. This method does not alter the original functionally-oriented implementation, but instead produces a description of a set of suggested objects and a structure (i.e., interactions) between those objects that are behaviorally-equivalent to the original implementation. Each object contains a set of components, some of which are visible in the interface of the object and others that are hidden within the object. These objects exhibit high cohesiveness between separate functions and between data and the functions that access the data. The basic premise of this method is that the functions (i.e., components) of the functionally-oriented implementation which operate on a given data structure define a reasonable object in an object-based orientation. This object-based design merely **suggests** objects since any subsequent use or implementation of the objects is open to further domain expert analysis and resolution of any implementation issues.

This method, however, will not perform reengineering miracles. The basic rule of Garbage-In, Garbage-Out applies. If the original analyzed code is made up of poorly designed functions and data stores, then this method will attempt to form objects based on the commonalities of this poor use. Therefore, it is a basic assumption of this method that some care has been given in the original functionally-oriented implementation to create meaningful functions and to carefully control accesses to information.

This method is made up of three primary activities: Perform Initial Analysis, Define Candidate Objects, and Integrate Objects (see Figure 1). The primary input to this method is the existing functionally-oriented implementation. Secondary inputs to this method, in the form of data or expertise, help to control the formation of objects. The output from this method are suggested objects. These objects can, for example, form the basis of a reuse library or can be implemented in an object-oriented programming language. As shown in Figure 1, this output is partitioned into two outputs: Unintegrated Objects and Integrated Objects. Most implementations contain some partitioning that was important to the original designers (i.e., functional subsystems) or to management (i.e., partitioning work assignments). These partitionings are referred to in this method as *subsystems*. When forming the Unintegrated Objects output, this method retains this subsystem organization during the formation of objects. The formation of the Integrated Objects output removes these subsystem boundaries by integrating all of the Unintegrated Objects. The result is a system view of the objects. These two object-based views of the functionally-oriented implementation are presented so that a choice can be made between the two views since there may be project-specific reasons that the existing structure (i.e., the subsystem boundaries) must be retained. There is no requirement in this method that this final activity need be performed. However, nothing is lost since both integrated and non-integrated object sets are produced. If you know that you want (or your organization wants) to retain the subsystem organization, then the Integrate Objects activity can be viewed as optional.

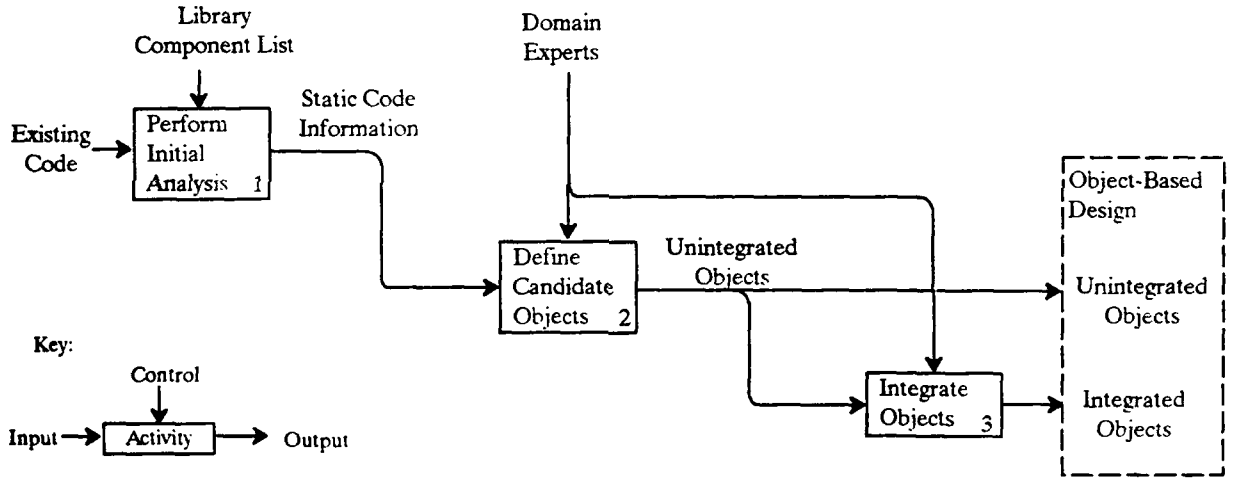



Figure 1. Method Information Flow

Tools and automation can be used for performing the activities of this method. All aspects of this method can be applied manually; however, this may prove to be very tedious. Static analysis tools can be used to extract the *context* and *dependency* information from existing code. Likewise, the tests and subactivities of the latter two activities can be automated to assist the user in performing this method. The application of automation to this method is discussed further in Section 4.

### 2.1.1 Perform Initial Analysis

The Perform Initial Analysis activity is the initial step in this method. It is an analysis step that obtains the information needed for forming objects. The existing functionally-oriented implementation is the input to this step. The primary action taking place in this activity is the creation of context and dependency sets for each component. These sets are determined by analyzing for each function the data accessed and the functions called. A component is said to be in the context of a given component if it

calls or accesses the given component. A component is said to be in the dependency set of a given component if the given component calls or accesses it. Also included in this activity is an enumeration of the *library components* of the system. Library components are functions within the functionally-oriented implementation that are at too low of a level (e.g., a programming language's run-time library components) to warrant being included in the formation of objects. Library components are identified as such to prevent them from becoming the basis for the formation of objects.

### 2.1.2 Define Candidate Objects

The second step in this method is the Define Candidate Objects activity. This activity uses the information gathered in the previous step to begin the formation of *candidate objects*. Objects formed by this method are called candidate objects because all objects are subject to review and approval by domain experts. This activity focuses on one subsystem at a time. A single subsystem is evaluated with no knowledge of the other subsystems in the implementation. Domain experts or persons knowledgeable in the functionally-oriented implementation will be required to identify these subsystem boundaries.

Initially, all subsystem components are *unallocated*, which means that they are not yet assigned to any particular candidate object. Candidate objects are formed by assigning the unallocated components to existing candidate objects. Candidate objects are basically formed whenever components share sufficient commonality in their dependency sets. The candidate objects possess *interface components* and *hidden components*. The interface components reveal only those aspects of the object that need to be known outside the object. The hidden components include any data that the interface components manipulate along with any internally hidden components that do not need to be exported.

This activity is repeated until all components are allocated to candidate objects. At this point, domain experts should review the candidate objects to determine if they reflect the cohesiveness expected for the given system. Domain expert reviews are, by their nature, very subjective processes which rely entirely upon the intuitive sense of the domain expert. This activity results in an object-based design that retains the subsystem boundaries present in the original functionally-oriented implementation.

### 2.1.3 Integrate Objects

The Integrate Objects activity is the final step in this method. It takes the unintegrated components from the previous activity as input and *integrates* all of the candidate objects from the individual subsystems into a single set of candidate objects. Essentially, this activity removes all of the subsystem boundaries and takes an overall system view to the formation of candidate objects. Sometimes separate candidate objects are merged into single objects. Sometimes candidate objects are split into separate pieces based on new information available from performing the integration. Splitting candidate objects is based on the fact that some components can access other subsystems components. The integration of such information can shed new light on how candidate objects are formed and, as such, can cause some previously formed candidate objects to be broken apart. At this point, domain experts should once again review the candidate objects to determine if they reflect the cohesiveness expected for the given system. This activity results in an object-based design that removes the subsystem boundaries present in the original functionally-oriented implementation.

## 2.2 INITIAL ANALYSIS

The Initial Analysis activity extracts the information needed from the functionally-oriented implementation to form objects. This activity, illustrated in Figure 2, initially creates the context and

dependency structures which reflect the static interactions in the code, and then identifies the library components of the implementation.

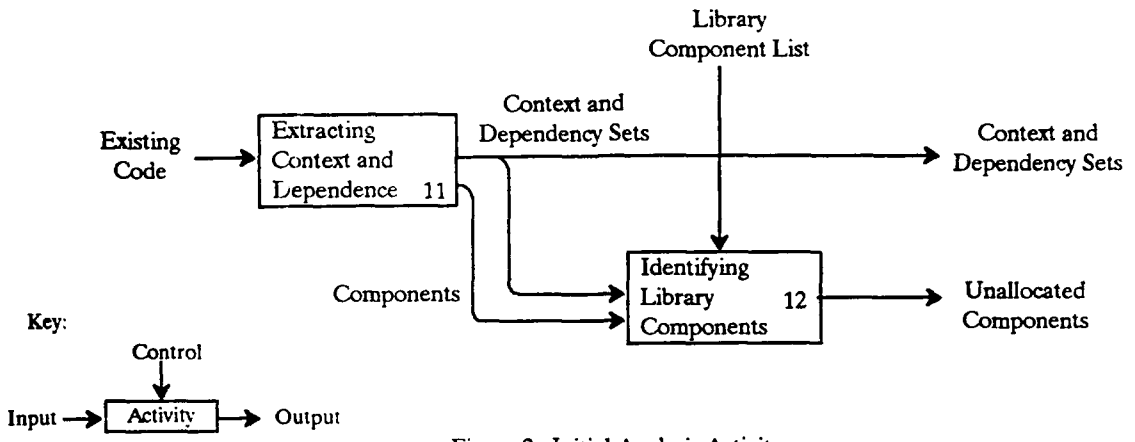



Figure 2. Initial Analysis Activity

### 2.2.1 Extracting Context and Dependence

A functionally-oriented implementation is made up of components such as functions, procedures, and data acted upon by these other components. Interactions between these components can take the form of calls and accesses. Static code information (i.e., information regarding the code's structure; not its run-time execution) is the basis for the formation of candidate objects in this method. This static code information needs to contain the following information for each component, C, in the functionally-oriented implementation:

- The set of all components called or accessed by C. The set can include *functional components* and *data components*. Typical examples of functional components include functions and procedures. Typical examples of data components include variables, records, and arrays. If C is a data component, then this set will be empty since data components do not call other components.

- A set of all functional components that call or access C.

This static code information is organized into two complementary structures: context and dependency sets. Each component in the functionally-oriented implementation has a context and dependency set, either of which may be empty.

The context of a component, denoted as a function *c-set*(component) which returns the context set, is the set of all components that call or access it. Both functional components and data components can have a context; the only exception would be the implementation's main program. In Figure 3, the context of component X is the set of components A, B, and C. The dependency set of a given component, denoted as a function *d-set*(component) which returns the dependency set, is the set of all functions or data components that it calls or accesses. Only functional components can actually have a dependency set since they are the only components that can call other components. Data components do not make calls to other components. As shown in Figure 3, the dependency of component X is the set of components Y and Z. The context and dependency functions are complementary in that if X $\in$ c-set(A), then A $\in$ d-set(X). These sets are established for each component in the functionally-oriented implementation.

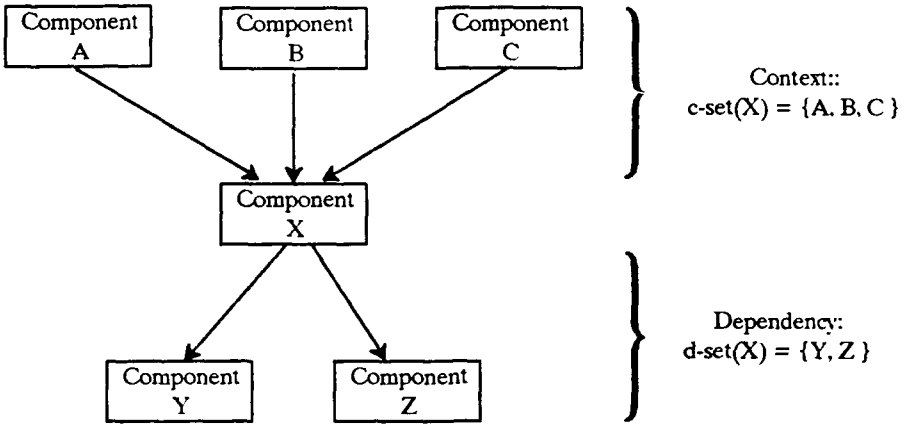



Figure 3. Context and Dependency of a Component

### 2.2.2 Identifying Library Components

The context and dependency sets, formed in Section 2.2.1, are the basis for forming objects in later activities. The object formation activities will analyze the context and dependency sets looking for sufficient commonality to form candidate objects. However, some functions are too fundamental or pervasive to include in the formation of objects. For example, if many of the components in a subsystem performed some type of printing, then these components would share the commonality of calling printing functions. It would likely be an error to group these components together simply because they all performed printing. Therefore, the printing functions can be identified as library components so that the formation of objects can focus on the more pertinent components in the implementation. All components designated as library components should be removed from the set of unallocated components for a given subsystem. In this manner, the library components will not be considered during the object formation activities.

## 2.3 DEFINING CANDIDATE OBJECTS

The Defining Candidate Objects activity uses the information obtained from the Initial Analysis activity to begin the process of forming objects. This activity is applied on a subsystem-by-subsystem basis. The premise here is that the subsystems were established by a project to reflect some project-specific or functional organization of the implementation. Therefore, this activity is performed on each subsystem separately in the overall implementation.

This activity, illustrated in Figure 4, allocates components to candidate objects. When this activity is initiated, all components are unallocated and no candidate objects are formed. Since components are only allocated to **known** candidate objects, an initial candidate object is formed. Unallocated components are then one-by-one either added to existing objects, combined with other components to form new objects, or are formed into new one-component objects. in that order. With the formation of each new candidate object, the base of known candidate objects is increased. This process iterates until all components have been allocated to candidate objects. Final analyses for finding hidden components and for review by a domain expert complete this activity. The candidate objects formed during this activity will contain a set of interface components, a potentially empty set of hidden components, and

a dependency set for the object. Once this activity is completed, the candidate objects are ready for integration (described in Section 2.4).

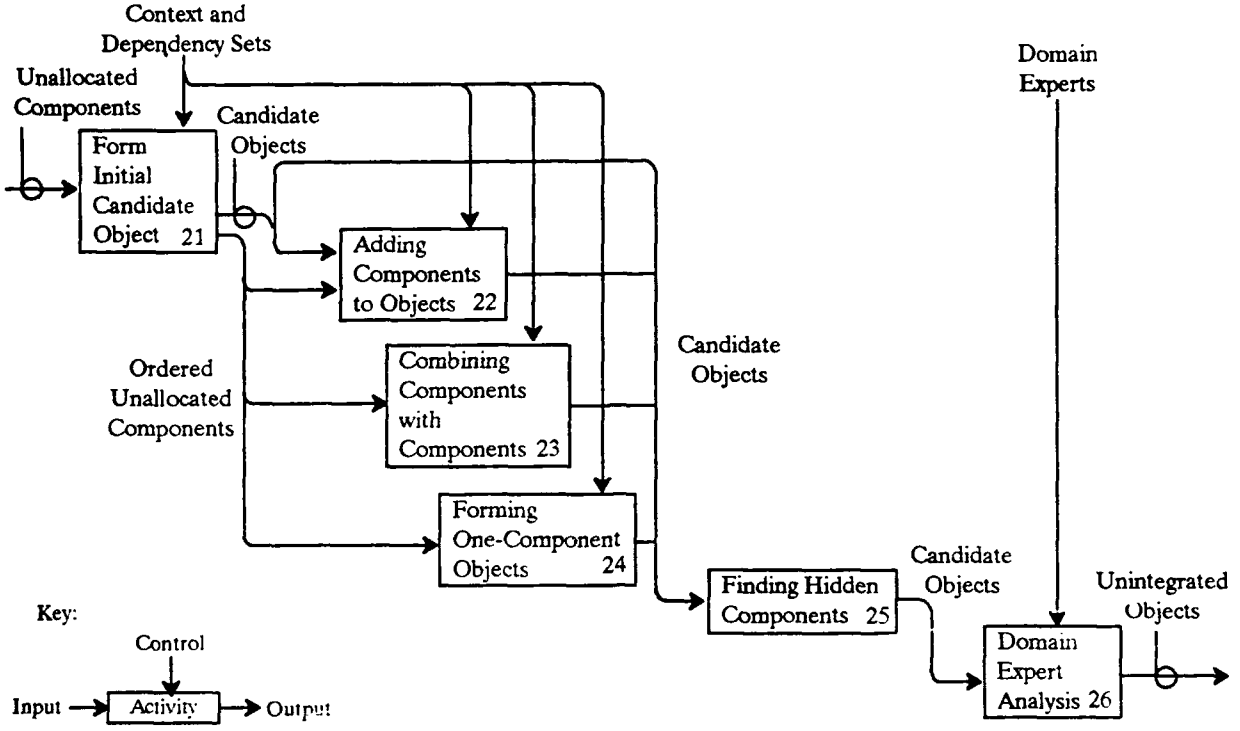



Figure 4. Defining Candidate Objects Activity

### 2.3.1 Forming the Initial Candidate Object

The subactivities of the Define Candidate Objects activity attempt to assign unallocated components to **known** candidate objects, which presumes that an initial candidate object is available. Therefore, the initial step in defining candidate objects is the formation of an initial candidate object for the subsystem.

To actually form the initial candidate object, perform the following steps:

1. Order all unallocated components of a subsystem by the size of their dependency sets (from largest to smallest). This ordering reflects the relative complexity of the subsystem's unallocated components and assumes a component calling many other functions or accessing lots of data is more complex than another component that calls fewer functions or accesses less data. Since it is used by the other subactivities of this activity, this ordered unallocated component set should be retained.
2. Choose the most complex component from the ordered unallocated component set. If there is a tie for the most complex unallocated component, then simply pick one using any arbitrary method (e.g., alphabetical ordering).
3. Form an initial candidate object with the selected component as its interface component.
4. Remove the component selected for the initial candidate object from the set of unallocated components since it is now assigned to a candidate object.

5. Set the initial candidate object's dependency set to the dependency set of the selected component.

### 2.3.2 Adding Components to Objects

This subactivity adds unallocated components to the interface of existing candidate objects (which the first time through is simply the initial candidate object). This subactivity tests the commonality between the dependency set of each unallocated component and the dependency sets of the candidate objects. If sufficient commonality exists, then the component is added to the candidate object's interface and the dependency sets are merged.

This subactivity is made up of two tests. These tests are applied to the most complex component (i.e., the one with the largest d-set) in the ordered set of unallocated components, and is attempted for each existing candidate object until successful. It is possible, however, that all attempts at assigning the unallocated component to the existing candidate objects will fail due to no commonality. In this situation, simply move on to the Combining Components with Components subactivity in Section 2.3.3.

Add unallocated component, U, to the interface of a candidate object, CO, if one of the following tests succeed (apply tests in order):

1. **d-set**(U) $\subset$ **d-set**(CO)

   This test states that if the dependency set of the unallocated component is a subset of the dependency set of the candidate object, then the test succeeds.

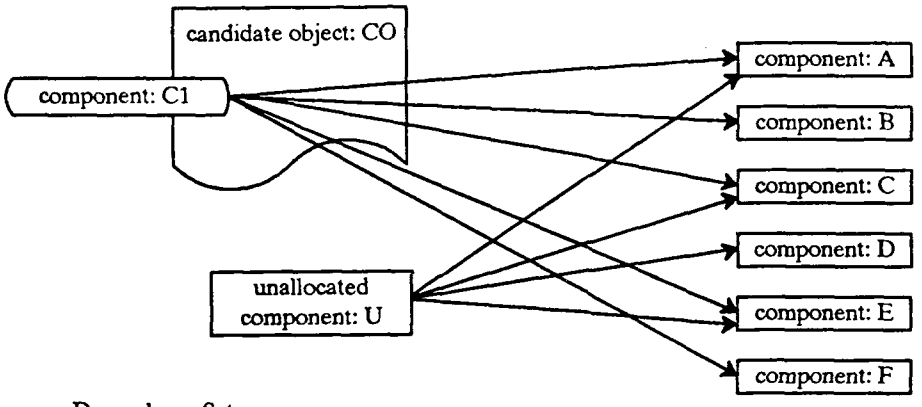
2. $|$**d-set**(CO) $\cap$ **d-set**(U)$| \geq$ ( **min** ($|$**d-set**(CO)$|$, $|$**d-set**(U)$|$) $*$ ***threshold*** )

   This test says to initially obtain the intersection of the dependency sets of the candidate object and the unallocated component. If the size of this intersection is greater than or equal to the size of the shorter of the two dependency sets **multiplied by** some thresholding factor, then the test succeeds. In other words, a component should be added to a candidate object if the commonality of the two dependency sets (i.e., the intersection) is greater than or equal to some percentage of the shorter of the two dependency sets (i.e., the **min** times the threshold). The threshold is present to tune this test for forming objects that are satisfactory to the domain experts. The larger the threshold, the more an exact match is needed to add the component to the candidate object (i.e., a threshold of 1.0 would require that the intersection be **equal** to the d-sets of either the candidate object or the unallocated component for success). The lower the threshold, the more easily components are added to the candidate objects. The threshold used in this test is independent of any other thresholds used in other activities.

Figure 5 (on page 12) illustrates these tests being applied to a candidate object, CO, and an unallocated component, U. In this example, the first test fails because there is not a subset between the dependency sets of the component and the candidate object. However, the second test allows the component to be added to the candidate object because sufficient commonality is present.

If the tests of this subactivity indicate that the unallocated component should be added to a candidate object, perform the following steps:

1. Add the selected component to the interface of the candidate object's interface component set.




Dependency Sets:

d-set(CO) = {A, B, C, E, F}
d-set(U) = {A, C, D, E}
d-set(CO) ∩ d-set(U) = {A, C, E}

Sizes:

|d-set(CO)| = 5
|d-set(U)| = 4
|d-set(CO) ∩ d-set(U)| = 3

Therefore, if the threshold is .65, then:

|d-set(CO) ∩ d-set(U)| ≥ ( **min** (|d-set(CO)|, |d-set(U)|) * threshold )
3 ≥ ( **min** (5, 4) * .65 )
3 ≥ (4 * .65) = (2.6)
True

*Figure 5. Adding Components to Objects*

2. Remove the component from the set of unallocated components since it is now assigned to a candidate object.

3. The dependency set for the candidate object remains unchanged unless the candidate object only contained a single component in its interface. In this special situation, the candidate object's dependency set is equal to the intersection of the dependency sets of the candidate object and the added component (i.e., d-set(CO)=(d-set(CO)∩ d-set(U)).

### 2.3.3 Combining Components with Components

This subactivity forms new candidate objects from two currently unallocated components that have commonality in their dependency sets. This subactivity tests the commonality between the dependency sets of unallocated component pairs. If sufficient commonality exists, then a new candidate object is formed with the two components as interface components and the dependency set of the new candidate object reflecting the commonality of the dependency sets of the two components. Whenever a new candidate object is formed using this subactivity, restart the Adding Components to Objects activity in Section 2.3.2. The method is restarted here since there is now an additional candidate object into which to add unallocated components.

This subactivity is made up of two tests. These tests are applied to the most complex component (i.e., the one with the largest d-set) in the ordered set of unallocated components. This most complex component is paired with each successively less complex component in the ordered set of unallocated

components. If for some reason this subactivity does not successfully find a suitable match for the most complex component, move on to the Forming One-Component Objects activity in Section 2.3.4.

Form a new candidate object encompassing the unallocated components, U1 and U2, if one of the following tests succeed (apply them in order):

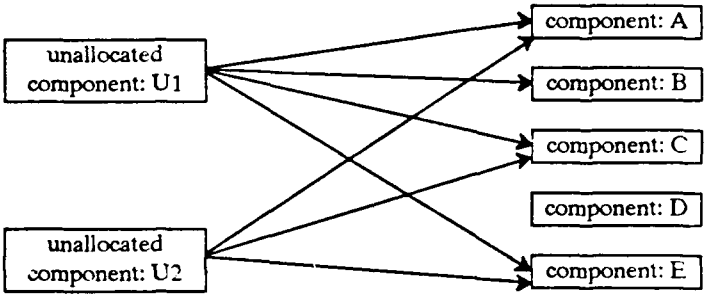
1. **d-set(U1) ⊂ d-set(U2) ∨ d-set(U2) ⊂ d-set(U1)**

   This test states that if the dependency set of either unallocated component is a subset of the other unallocated component's dependency set, then the test succeeds.

2. **|d-set(U1) ∩ d-set(U2)| ≥ (|d-set(U1)| * threshold ) ∨**
   **|d-set(U1) ∩ d-set(U2)| ≥ (|d-set(U2)| * threshold )**

   This test says to initially obtain the intersection of the dependency sets of the two unallocated components. If the size of this intersection is greater than the size of either of the individual dependency sets multiplied by some thresholding factor, then the test succeeds. In other words, two components should form a candidate object if the commonality of their dependency sets (i.e., the intersection) is greater than or equal to some percentage of one of the two dependency sets individually. Thresholding works the same here as in Section 2.3.2, except that different thresholding values can be used.

Figure 6 illustrates these tests being applied to two unallocated components, U1 and U2. In this example, the first test succeeds; therefore, the second test is unnecessary.




Dependency Sets:
d-set(U1) = {A, B, C, E}
d-set(U2) = {A, C, E}

Therefore:
d-set(U1) ⊂ d-set(U2) ∨ d-set(U2) ⊂ d-set(U1)
{A, B, C, E} ⊂ {A, C, E} ∨ {A, C, E} ⊂ {A, B, C, E}
(False ∨ True) = True

Figure 6. Combining Components with Components

If the tests of this subactivity indicate that two unallocated components should form a new candidate object, perform the following steps:

1. Add the two components to the interface of the new candidate object's interface component set.

2. Remove both components from the set of unallocated components since they are now assigned to a candidate object.

3. The dependency set for the candidate object depends upon which test above succeeded. If Test 1 succeeded, then the new object's dependency set becomes the larger of the dependency sets of the two components, i.e., d-set(CO)=d-set(U1), if |d-set(U1)| > |d-set(U2)|, else d-set(CO)=d-set(U2). If Test 2 succeeded, the new object's dependency set becomes the intersection of the dependency sets of the two components, i.e., d-set(CO)=d-set(U1)∩ d-set(U2).

### 2.3.4 Forming One-Component Objects

This subactivity forms new candidate objects when the previous subactivities have failed to assign the most complex unallocated component to an existing candidate object. Whenever a new candidate object is formed using this subactivity, restart the Adding Components to Objects activity in Section 2.3.2. The method is restarted here since there is now an additional candidate object into which to add unallocated components.

To form this one-component object, perform the following steps:

1. Form a candidate object with the unallocated component as its interface component.

2. Remove the component from the ordered set of unallocated components since it is now assigned to a candidate object.

3. The dependency set for the candidate object is set to the dependency set of the interface component, i.e., d-set(CO) = d-set(interface component).

### 2.3.5 Finding Hidden Components

After all unallocated components have been assigned to candidate objects, this subactivity determines if any of the interface components of the candidate objects can become hidden components. Hidden components are components of a candidate object that are not part of its interface and are only called or accessed by components within that candidate object. Many of the components contained in the one-component objects will likely become hidden components of another candidate object.

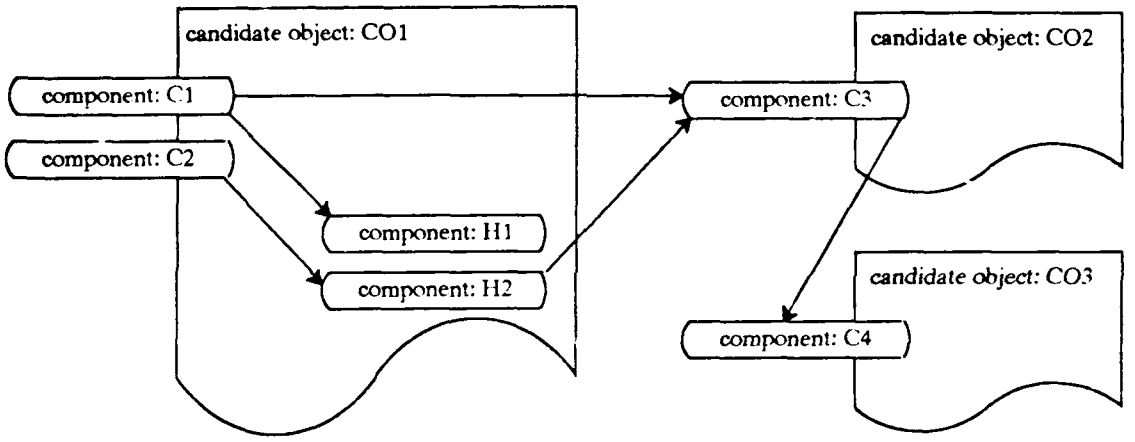
This subactivity consists of a single test that is applied to every interface component of every candidate object. For each component (referred to as C) considered in conjunction with each known candidate object (referred to as CO), the following test is applied:

c-set(C) ⊂ ( (set of all interface components of a given candidate object, CO) ∪
(set of all hidden components of a given candidate object, CO) )

The basic premise of this test is that if a given component (whether it is currently within a one-component object or within a multi-component object) is only accessed by the components (both interface and hidden) of a single candidate object, then the given component can become a hidden component of that candidate object. The component is removed from the interface of its original candidate object and made into a hidden component of the candidate object with which the test succeeded. If a candidate object becomes devoid of any components because of this subactivity, then the empty

candidate object is discarded since it is no longer needed. This subactivity is repetitive since placing any component into a candidate object (as a hidden component) increases the number of components contained in that candidate object, thus, increasing the chances for other components to become hidden components of this same candidate object. This subactivity repeats until there are no further changes in hidden components.

Figure 7 illustrates an example of this subactivity being applied to two candidate objects, CO1 and CO2. The candidate object, CO1, contains two interface components, C1 and C2, and two hidden components, H1 and H2. The candidate object, CO2, is a one-component object containing a single interface component, C3. This test succeeds since the context set of C3 is a subset of the set of components contained in CO1. C3 qualifies to become a hidden component of CO1. When this occurs, CO2 will become empty and can be discarded. The repetitive nature of this subactivity becomes evident after C3 has been added to CO1 as a hidden component. The component C4 of the one-component object CO3 will now also qualify as a hidden component of CO1 since its only context component is C3, which is now a hidden component of CO1.




From the above diagrams:
c-set(C3) = {C1, H2}
component set for CO1 = {C1, C2, H1, H2}

Therefore:
c-set(C) ⊂ ( (set of all interface components of a given candidate object, CO) ∪
(set of all hidden components of a given candidate object, CO) )
{C1, H2} ⊂ ( {C1, C2} ∪ {H1, H2} )
True

Figure 7. Finding Hidden Components

### 2.3.6 Domain Expert Analysis 1

After each subsystem is analyzed and a set of candidate objects are identified, a domain expert should examine the set of objects. The questions that the expert must ask about each candidate object are:

- Does the set of interface components form a logically-related portion of the implementation's behavior?

- Are the interface components likely to change together?
- Are the candidate objects of appropriate size (measured in total components)?

If the answers to these questions indicate that changes need to be made, then these changes should be well documented. The object-based design formed by this activity is simply a **recommendation** based on the static interactions between the functionally-oriented components. Expert experience, organizational standards, or even strong preference may call for altering the design. Be sure that all changes are documented and carefully considered.

If changes need to be made, because some of the components of a candidate object do not belong with the other components of the object, then the following actions can be taken:

- **Determine if new library components should exist.** Since objects are primarily constructed based on commonality between component dependency sets, review the dependency sets of the components in conflict within the object. It may be necessary to make some of the components in these dependency sets into library components so that they are not included in the formation of objects. Any time a new library component is identified, the activities of this method have to be restarted from the beginning.
- **Make the problem components into new one-component objects.** If there are no new library components that can be identified, then simply remove any problem components from the candidate objects in which the conflict occurred and make them into new one-component objects. After removing any component from an object, review each of the hidden components of the candidate object from which the component was removed. Some of the hidden components may no longer qualify as hidden components. They were originally made into hidden components because they were only called or accessed by other components within the candidate object. However, after removing any problem component, this condition may no longer be true. If a hidden component must also be removed, the particular hidden component is removed from the candidate object and a new one-component object is formed around this previously hidden component. After removing any hidden component from a candidate object, you need to reconsider all other hidden components in the candidate object since part of their context may have just been removed. Unlike identifying new library components above, forming these new one-component objects does not warrant restarting the entire method. After the domain expert is satisfied with these changes, move on to the Integrating Objects activity in Section 2.4.

## 2.4 INTEGRATING OBJECTS

Integrating Objects, the final activity, removes all of the subsystem boundaries imposed by the functionally-oriented implementation. This activity integrates subsystems, one-at-a-time, into a *compound subsystem*. All integration is nondestructive, however. As each subsystem is integrated, copies should be made of the subsystems as they are integrated, thus preserving the pre-integration candidate objects and forming a new set of post-integration objects. In this way, a domain expert can look at the object definitions at both the system and the subsystem levels to determine which seem more appropriate for the given implementation. Subsystems are integrated in order of decreasing complexity: the most complex subsystem is the one with the greatest number of components (both interface and hidden). The most complex subsystem will serve as the initial compound subsystem. All other subsystems will be integrated into this compound subsystem.

As each individual subsystem is integrated into the compound subsystem, the context and dependency information for all components in the subsystems are integrated, as illustrated in Figure 8 .Component integration potentially increases the context and dependency sets for the components of the objects being integrated. After this component information is integrated, then the objects are integrated which attempts to merge any of the existing candidate objects into single objects. These two steps are repeated as each subsystem is integrated into the compound subsystem. A review by a domain expert completes this activity.

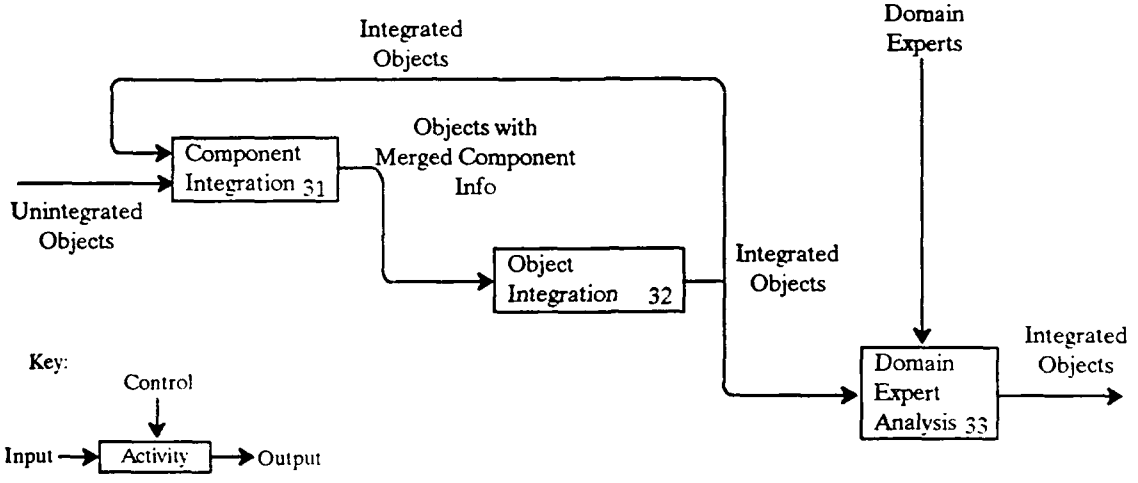



Figure 8. Integrating Objects Activity

### 2.4.1 Component Integration

As each subsystem is integrated into the compound subsystem, the components (both interface and hidden) of the subsystem and the compound subsystem are initially reviewed to determine if integration will cause changes to the interface or hidden components of either subsystem. This subactivity integrates the context and dependency information for each component of the objects, and then searches to see if this updated information causes any hidden components to become interface components.

#### 2.4.1.1 Integrating the Component Information

The partitioning of the original functionally-oriented implementation into subsystems most likely causes the complete definition of some components to be incomplete. This is because some components may be defined in one subsystem and accessed by another subsystem. In this situation, the complete context and dependency set for the component will not be completely known until both subsystems are reviewed together (i.e., component integration).

When you integrate a subsystem into the compound subsystem, all common components must have their component information integrated. The integration of the component's context and dependency sets is simply the union of each of the separate sets. Figure 9 illustrates the integration for a component, C1, that is present in two separate subsystems. In Subsystem 1, C1 is defined, and in the compound subsystem, C1 is called by B and C and has no dependency set. Integrating Subsystem 1 into the compound subsystem causes C1's context and dependency sets to become integrated. The revised

information (i.e., the integrated component information) is updated in Subsystem 1 **and** the compound subsystem.

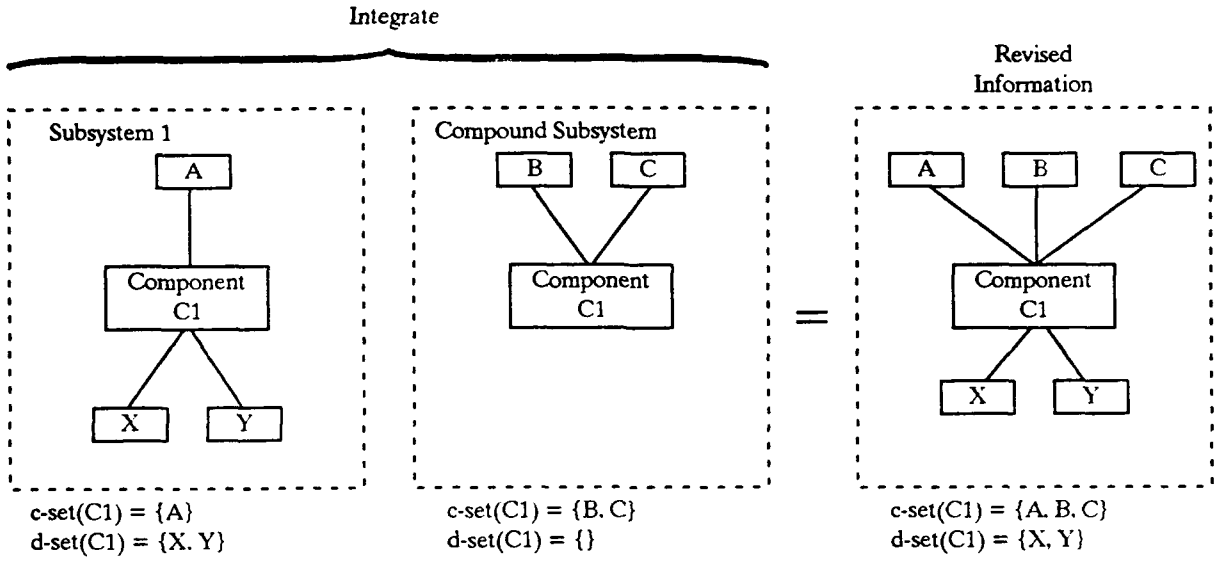



Figure 9. Component Integration Example

### 2.4.1.2 Making Hidden Components into Interface Components

After the component information is integrated, some of the hidden components may no longer qualify as hidden components. They were originally made into hidden components because they were only called or accessed by other components within a single candidate object. However, after integration, this condition may no longer be true. In this situation, the particular hidden component is removed from the candidate object and a new one-component object is formed around the previously hidden component. After removing any hidden component from a candidate object, you need to reconsider all other hidden components in the candidate object since part of their context may have just been removed.

In Figure 9, component C1's context set was expanded due to the integration of Subsystem 1 and the compound subsystem. In the compound subsystem, this integration expanded C1's context set from {B, C} to {A, B, C}. Therefore, if C1 is a hidden component in the resulting compound subsystem, then you need to check to see if it should become an interface component instead. If C1's context is split between two separate objects within the compound subsystem (i.e., A is in one object and B and C are in another), then C1 will not be able to remain as a hidden object since it is needed by two different objects. You will need to remove it from its current object and place it into a one-component object with C1 as the interface component.

## 2.4.2 Object Integration

After completing the component integration, integrate the objects of the subsystems. This object integration moves all objects from the subsystem being integrated into the compound subsystem. It then determines whether any of the objects can be merged with other objects of the compound

subsystem. This activity initially integrates objects, and then it applies several checks to determine the resulting legality of the hidden and visible components of the integrated compound subsystem. Any discrepancies are corrected.

#### 2.4.2.1 Integrating the Candidate Object Information

This subactivity consists of two tests for integrating objects within the compound subsystem: the Distributed Object Test and the Aggregate Object Test. The Distributed Object Test attempts to locate objects whose definition was partially distributed across two of the integrated subsystems. It examines the interface components of both objects and their dependency sets to determine if they can actually form a logically related object. The Aggregate Object Test checks the component sets of two objects to determine if one is a subset of the other. If so, it is likely that the two objects can be merged into a single object.

- ***Distributed Object Test.*** The Distributed Object Test operates on the dependency sets of any pair of objects in the compound subsystem, say CO1 and CO2, and is described as:

  $$\text{d-set(CO1)} \subset \text{d-set(CO2)}$$

  This test states that if the dependency set of any candidate object is a subset of any other candidate object's dependency set, then the test succeeds.

  If this test indicates that candidate objects should be merged into a new candidate object, then perform the following steps:

  1. Merge the components (both interface and hidden) from the two candidate objects into the new candidate object.
  2. Set the dependency set of the newly merged candidate object to the intersection of the dependency sets of the two merged objects, i.e., d-set(new-object) = d-set(CO1)$\cap$d-set(CO2).

- ***Aggregate Object Test.*** The Aggregate Object Test operates on the set of components (both interface and hidden) of objects in the compound subsystem, say CO1 and CO2, and is described as:

  $$\text{component-set(CO1)} \subset \text{component-set(CO2)}$$

  This test states that if the components of any candidate object are a subset of any other candidate object's components, then the test succeeds. This essentially means that the one object is completely subsumed within the second object.

  If this test indicates that candidate objects should be merged into a new candidate object, then perform the following steps:

  1. Merge the components (both interface and hidden) from the two candidate objects into the new candidate object. The redundant components should only appear once in the merged candidate object.
  2. Set the dependency set of the newly merged candidate object to the intersection of the dependency sets of the two merged objects, i.e., d-set(new-object)=d-set(CO1)$\cap$d-set(CO2).

After these tests have been applied, the following checks must be performed to ensure that a legal set of objects results from the object integration:

1. Check each of the interface components of the merged candidate object to determine if they can still remain in the candidate object with the object's new dependency set. This check is identical to the Adding Components to Objects activity in Section 2.3.2 for placing components into the interface of a candidate object and consists of two tests. The first test (adapted for this integration step) states that if the dependency set of an interface component I is a subset of the dependency set of the candidate object CO, i.e., d-set(I) $\subset$ d-set(CO), then the interface can remain in the candidate object. Also, if this first test fails, then obtain the intersection of the dependency sets of the candidate object and the interface component. If the size of this intersection is greater than or equal to the size of the shorter of the two dependency sets **multiplied by** some thresholding factor, i.e., $|\text{d-set(CO)} \cap \text{d-set(I)}| \geq (\textbf{min}\ (|\text{d-set(CO)}|, |\text{d-set(I)}|) * \text{threshold}$, then the interface can remain in the candidate object.

   If any of the interface components fail this check, then the interface component must be removed from the new candidate object and placed into a new one-component object.

2. If any interface components of a candidate object were removed because of failing the above dependency set check, then the hidden components of the candidate object must also be reconsidered. One of these hidden components may have contained the removed interface component in its context set (i.e., the interface component called or accessed the hidden component). In this situation, the hidden component is removed from the candidate object and a new one-component object is formed around this previously hidden component. After removing any hidden component from a candidate object, you need to reconsider all other hidden components in the candidate object since part of their context may have just been removed.

3. Duplicate components also need to be resolved. As objects from different subsystems are integrated, there is the possibility that a component will be duplicated (i.e., a component might have been an interface component in one candidate object while being a hidden component in another candidate object). Since the context and dependency information for the duplicate components are identical (because of Section 2.4.1.1 on integrating component information), one of the components can be discarded. The following rules apply to removing duplicate components:

   a. If a component is present in a one-component object and also present in another multi-component object, then discard the one-component object and its contents.

   b. If a component is present in two one-component objects, then discard one of the one-component objects and its contents.

   c. If a component is in the interface of one object and is hidden in another object, then discard the hidden component.

   d. If a component is hidden in two objects, then make one of the hidden components into a new one-component object (removing it from its containing object) and discard the other hidden component. When any new one-component object is formed due to this rule, reapply rule 2 above.

#### 2.4.2.2 Reexamining One-Component Objects for Hidden Components

This subactivity revisits each one-component object in the integrated compound subsystem to determine if any interface components of these objects can become hidden components in any other object. This subactivity consists of a single test that is applied to every one-component object. For each one-component object (which we will refer to C) considered in conjunction with each known candidate object (referred to as CO), the following test is applied:

$$\text{c-set}(C) \subset \big(\text{(set of all interface components of a given candidate object, CO)} \cup \text{(set of all hidden components of a given candidate object, CO)}\big)$$

This subactivity is identical to the Find Hidden Components activity in Section 2.3.5. The basic premise is that if a one-component object's interface component is only accessed by the components (both interface and hidden) of a single candidate object, then the given component can become a hidden component of that candidate object. The component is made into a hidden component of the candidate object with which the test succeeded, and the one-component object is discarded since it is no longer needed. This subactivity is repetitive since placing any component into an candidate object (as a hidden component) increases the number of components contained in that candidate object, thus, increasing the chances for other components to become hidden components of this same object.

### 2.4.3 Domain Expert Analysis 2

After each subsystem is integrated into the compound subsystem, a domain expert should examine the set of objects. The questions that the expert must ask about each candidate object are:

- Does the set of interface components form a logically-related portion of the implementation's behavior?
- Are the interface components likely to change together?
- Are the candidate objects of appropriate size (measured in total components)?

If the answers to these questions indicate that changes need to be made, then these changes should be well documented. The object-based design formed by this activity is simply a **recommendation** based on the static interactions between the functionally-oriented components. Expert experience, organizational standards, or even strong preference may call for altering the objects. Be sure that all changes are documented and are carefully considered.

If changes need to be made, because some of the components of a candidate object do not belong with the other components of the object, then the following actions can be taken:

- ***Determine if new library components should exist.*** Since objects are primarily constructed based on commonality between component dependency sets, review the dependency sets of the components in conflict within the object. It may be necessary to make some of the components in these dependency sets into library components so that they are not included in the formation of objects. Any time a new library component is identified, the activities of this method have to be restarted from the beginning.
- ***Make the problem components into new one-component objects.*** If there are no new library components that can be identified, then simply remove any problem components from the

candidate objects in which the conflict occurred and make them into new one-component objects. After removing any component from an object, review each of the hidden components of the candidate object from which the component was removed. Some of the hidden components may no longer qualify as hidden components. They were originally made into hidden components because they were only called or accessed by other components within the candidate object. However, after removing any problem component, this condition may no longer be true. If a hidden component must also be removed, the particular hidden component is removed from the candidate object and a new one-component object is formed around this previously hidden component. After removing any hidden component from a candidate object, you need to reconsider all other hidden components in the candidate object since part of their context may have just been removed.

Unlike identifying new library components above, forming these new one-component objects does not warrant restarting the entire method. After the domain expert is satisfied with these changes, integration of additional subsystems can continue.

After the completion of all subsystem integration, the domain expert should review the final integrated candidate objects against the non-integrated candidate objects produced in Section 2.3. This review may cause the domain expert to reconsider the boundaries imposed by the original functionally-oriented implementation.

## 2.5 USING THE RESULTS

This method identifies an object-based design as a set of candidate objects in both integrated and unintegrated forms. These objects reflect the desirable object-based characteristics of encapsulation, information hiding, and problem-space orientation. These characteristics should provide a basis for addressing the reasons given in Section 1.3 regarding **why** there might be a need to reengineer functionally-oriented implementations.

By providing well-defined interfaces and hidden information, you can easily implement these objects in object-oriented languages or in languages that support (or enforce) information hiding. These objects can also form the basis for libraries of reusable components.

Even though, in its current form, this method does not directly support the transformation of the object-based design into particular languages or object-oriented paradigms, the documentation provided by this method still provides an excellent basis for performing these transformations.

# 3. METHOD VALIDATION

The Consortium has validated this method on a pilot project with the SYSCON Corporation of Williamsburg, Virginia. The pilot project applied this method to an implementation of an Automatic Identification System, named SID. SID is a "smart card identification system" to control access to secure buildings, rooms, and equipment. This system is composed of multiple subsystems, hundreds of functions and data structures, and thousands of lines of code.

Aside from validating this method, the purpose for applying it to SYSCON's domain was to:

- Improve the maintainability and adaptability of the system.
- Produce a library of reusable software objects for future development.

The SID application was written in C, and the intent was to use this method to help transform the system to a C++ implementation.

This method was applied three separate times during the pilot project. The first application was performed manually to a subset of the SID application. No automation was involved. On the basis of this first application of the process, certain activities were refined (i.e., altered, added, deleted) to more accurately address the SYSCON application. The second application was performed on the entire system with some automation (see Section 4). The result was similar to the first application. New activities were added, some old ones were refined, and tailoring parameters were slightly altered. No activities were removed this time, however. The third application resulted in no alteration to the method and was done with automated assistance.

SYSCON's response to the result of applying the method has been very favorable. The objects defined capture related functions and data that can be reused in future development. The size of the objects are large enough so as to increase the productivity of reusing them over simply reusing their component parts. Also, the savings in man-hours by having automated support over having to evaluate the code "by hand" made the transformation to using this method reasonably cost-effective and salable to management.

Further validation of this method is necessary to identify whether this method is applicable to systems differing from SYSCON's. Tailoring occurred between applications of this method to obtain the desired results for SYSCON. Further experimentation is needed to identify exactly how tailoring is to be accomplished. The method can be applied, in its current form, to any system. However, it is likely that project-specific or company-specific tailoring will be required. Until further exploration occurs, the user will have to perform this tailoring. By altering this method, the resulting objects may improve over those created from using this method verbatim. The basic premises of this method, though, should hold true regardless of the tailoring performed.

*This page intentionally left blank.*

# 4. METHOD SUPPORT

This method is mechanical and tedious in nature. As such, the Consortium recommends using some form of automated support to perform the activities of the method. The extracting context and dependence portion of this method can likely be supported by commercial-off-the-shelf (COTS) products designed to extract the necessary information from implementations written in particular programming languages. Automated support for the rest of this method is currently not supported by any COTS products. However, the mechanical nature of the activities present in this method lends it to greatly benefitting from automated support. The method, as presented in this report, was practiced with SYSCON in manual and automated form. The manual form was tedious, but doable. Once automated support was present, however, productivity was greatly enhanced and error rates dramatically decreased.

During the pilot project between the Consortium and the SYSCON Corporation, this method was supported by several forms of automation. The static analysis of the C code was supported by a PC product named CDOC. CDOC extracted the calls and called-by relationships between functions and the access information for data structures. The rest of this method was supported by a rudimentary prototype called C2C$^{++}$. This C2C$^{++}$ prototype was written in Common LISP and utilized the Common LISP Object System (CLOS) to manage the objects created by this method. The C2C$^{++}$ prototype consisted of approximately 4700 lines of Common LISP code and was written to execute on a Sun 4 workstation (however, the prototype should be able to run on any platform that supplies a full Common LISP implementation).

The C2C$^{++}$ prototype primarily supports the activities and checks of the conceptual method. Candidate objects are created, analyzed, and removed based on the application of these activities and checks. The prototype, however, does not support the expert analysis or transforming the candidate objects to C$^{++}$. In its current form, any alterations desired by the domain expert to the candidate objects must be affected through Common LISP programming and CLOS interactions; the prototype does not currently handle making these changes to the candidate object base. The output of this prototype is a listing describing both integrated and non-integrated candidate objects and each of the components contained in these candidate objects.

*This page intentionally left blank.*

# 5. CONCLUSIONS

## 5.1 FINAL CONCLUSIONS

This method shows good potential for extracting an object-based design from functionally-oriented implementations. This method analyzes existing code and identifies a set of objects that are behaviorally-equivalent to the original functionally-oriented implementation. The resulting objects are a *cohesive* grouping of the original functions along with the data that those functions manipulate.

This method also appears to address the conventional motivations for reengineering existing systems. It aids the user in creating objects that exhibit proper encapsulation and information hiding characteristics using the information extracted from the functionally-oriented implementation. This method helps the user move from a functionally-oriented implementation to one that is object-based, and it provides a mechanical way of analyzing existing code to obtain this object-based viewpoint of the original implementation.

This method was validated on a pilot project with the SYSCON Corporation. The feedback from SYSCON has been very positive. SYSCON was able to help the Consortium in refining the method.

Finally, this method is preliminary and needs further exploration. This report documents the method's present state (as practiced on the SYSCON pilot project) so that interested technologists and methodologists can continue the exploration into transforming functionally-oriented implementations into object-based implementations.

## 5.2 FUTURE WORK

This method is preliminary. This section enumerates some of the future work that could make it more useful. The following are a list of some of these suggestions:

- Exploration into tailoring this method for a particular project or company. Currently, this tailoring is not well understood. Some activities are currently tailorable, as in the case with the thresholding values used in the activities. However, tailoring beyond these thresholds will require further exploration.

- Automated support would be very beneficial. Currently the activities and checks are only automated in prototype form. No support is provided for the expert analyses or the transformation to an object-oriented programming language. Automation, in the form of commercialization, would greatly enhance the usability of this method.

- Currently, lists of components and objects are ordered based on "complexity." This complexity is defined as components with the greatest number of dependencies or objects with

the greatest number of components. It is unclear if there are other ordering mechanisms that would benefit this method (i.e., new ways of selecting the initial candidate object within a subsystem). This could use some exploration.

- This method currently only operates on function-to-function and function-to-data interactions within the functionally-oriented implementation. However, future exploration could determine whether similar data-to-data interactions (e.g., the `typedef` statement in the C language) could be added as an additional basis for object formation.

# GLOSSARY

| | |
|---|---|
| C-set | A context set for a component. |
| Candidate objects | Objects created by this method for subsequent review and approval by domain experts. |
| Cohesiveness | The degree to which the tasks performed by a single program module are functionally related. (IEEE 1983) |
| Commonality | Common calls or data accesses between two components. |
| Components | Functional and data components from the original functionally-oriented implementation. |
| Compound subsystem | A object formed by the integration of the original functionally-oriented subsystem objects. |
| Context | For a given component, the set of all components that call or access it. Denoted as a c-set (component). |
| D-set | A dependency set for a component. |
| Data component | Any form of state or data storage in a system. Examples include variables, records, and arrays. |
| Data elements | The data structures (i.e., variables, records) and state variables maintained by a program. |
| Dependency | For a given component, the set of all functions or data components that it calls or accesses. Denoted as a d-set (component). |
| Encapsulation | The technique of isolating a system function within a module and providing a precise specification for the module. See also information hiding. (IEEE 1983) |
| Functional components | A unit of executable code in a system. Typical examples include functions and procedures. |

| | |
|---|---|
| Functionally-oriented | A system whose "boundaries of modules have been defined in a way that depends on the decomposition, which in turn depends on the functional characteristics of the specific application." (Graham 1991) |
| Functions | Code that is invoked by a calling statement. Functions include procedures, functions, and subroutines in programs. |
| Hidden component | A component assigned to a candidate object, but is not included in the object's interface. It is a non-exported component of the object. |
| Information hiding | The technique of encapsulating software design decisions in modules in such a way that the module's interfaces reveal as little as possible about the module's inner workings; thus, each module is a "black box" to the other modules in the system. The discipline of information hiding forbids the use of information about a module that is not in the module's interface specification. (IEEE 1983) |
| Integration | The formation of candidate components based on removing all of the subsystem boundaries taken from the original functionally-oriented implementation. Integration takes an overall system view to the formation of candidate objects. |
| Interface component | A component assigned to a candidate object, and is included in the object's interface. It is an exported component of the object. |
| Library components | Library components are functions within the functionally-oriented implementation that are at too low of a level to warrant being included in the formation of objects. Library components are denoted as such to prevent them from becoming the basis of the formation of objects. |
| Object | Cohesive, logically-related groups of functions and data that is manipulated by these functions. Objects contain an interface and its hidden information. The interface reveals those aspects of the object that need to be known outside the object. The hidden information include any data that the interface functions manipulate along with any internally hidden functions that do not need to be exported. |

| | |
|---|---|
| Object-based design | A description of a set of suggested objects and a structure (i.e., interactions) between those objects that is behaviorally-equivalent to the original implementation. The resulting design focuses on encapsulation, information hiding, and problem-space orientation. |
| Reengineering | The process of extracting valuable information from code to better understand the code or to help to improve the code. |
| Subsystem | A partitioning of a system. Subsystems can be formed based on logical groupings of related software and hardware. Subsystems can also be formed based on managerial work assignments. |
| Threshold | A factor used to control the effect of activities. |
| Unallocated components | Components that have not yet been assigned to any particular object. |

*This page intentionally left blank.*

# REFERENCES


Graham, Ian
1991

*Object Oriented Methods.* Reading, Massachusetts: Addison-Wesley Publishing Company.

IEEE
1983

*IEEE Standard Glossary of Software Engineering Terminology.* New York, New York: The Institute of Electrical and Electronics Engineers, Inc.

*This page intentionally left blank.*

# BIBLIOGRAPHY

Pole, Thomas. *Transitioning to the Object Oriented Software Development Paradigm Using C2C++: Recovering the Implicit Reusable Objects from a Non-Object Oriented Implementation,* Internal Working Paper. Herndon, Virginia: Software Productivity Consortium, May 1992.

*This page intentionally left blank.*

# PRODUCT USE SURVEY

**After reviewing and/or using our product, please tell us what you think by completing this short survey form.**

**Product name:** Preliminary Report on Extracting Object-Based Design from Functionally-Oriented Implementations

### How did you become aware of our product?

- ☐ Received information from company distribution point
- ☐ Heard a Consortium technical presentation
- ☐ Read about it in the Consortium *Quarterly*
- ☐ Saw information on a bulletin board
- ☐ Read article in a journal/trade publication
- ☐ Heard about it at a (non-Consortium) conference
- ☐ A fellow engineer told me
- ☐ Other _____

### Compare our product to competing products you are familiar with:

Mark the appropriate box with an X for our product. ☒

Circle the box to rate the competing product. ◯

| | Adequate | | | | Inadequate | | |
|---|---|---|---|---|---|---|---|
| | Totally | Very | Barely | Borderline | Somewhat | Mostly | Totally |
| Clarity | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| Organization | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| Usefulness | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| Presentation | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| Extensibility/modularity | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| Applicability to your project | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| OVERALL RATING | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

Name of competing product: _____

Source of guidance (guidebook, class, seminar, etc.): _____

### In your opinion, what are the two most important improvements we should make to our product?

_____

_____

### Do you believe our product helped you save time and effort compared to the competition? If so, how much?

| No | 1–2% | 2–5% | 5–10% | 10–20% | >20% | Don't Know |
|---|---|---|---|---|---|---|
| ☐ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

### On what kind of project will you use/have you used our product?

| Evaluation | IR&D | Proposal | Contract | Internal standard |
|---|---|---|---|---|
| ☐ | ☐ | ☐ | ☐ | ☐ |

### On what kind of problem will you use our product?

_____

_____

### How soon will you use our product?

| Immediately | Within 3 months | Within 6 months | When our proposal wins | Waiting on the RFP |
|---|---|---|---|---|
| ☐ | ☐ | ☐ | ☐ | ☐ |

### Estimated number of people in your project:

| 5 to 20 | 20 to 50 | 50 to 100 | 100 to 200 | More than 250 |
|---|---|---|---|---|
| ☐ | ☐ | ☐ | ☐ | ☐ |

### Your primary responsibility on this project has been:

| Project management | Systems requirements design | Systems design | Software requirements analysis | Software design | Testing |
|---|---|---|---|---|---|
| ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

☐ Other _____

SPC Form 750–2, 4/92

Please return this survey to:

The Software Productivity Consortium
SPC Building
2214 Rock Hill Rd.
Herndon, VA 22070–4005

If you have any questions,
please call the Technology Transfer Clearinghouse
(703)742–7211

SOFTWARE PRODUCTIVITY CONSORTIUM

# PRODUCT USE SURVEY

NAME _____ PHONE _____

TITLE _____ MAIL STOP _____

DIVISION _____

COMPANY _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

FOLD HERE

**GIVE US YOUR OPINION ABOUT OUR PRODUCT.**

**Please complete the survey form above and on the reverse side.**

**FOLD, TAPE, AND MAIL.**

We are working continually to improve our products. After you have reviewed or used our product, give us your opinion by completing the short survey above and on the reverse side. It should take you only a couple of minutes. We will send you the new Software Productivity Consortium poster in return.

Thank you,

The Software Productivity Consortium

FOLD HERE

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS MAIL PERMIT NO. 7702 HERNDON VA

POSTAGE WILL BE PAID BY ADDRESSEE

TECHNOLOGY TRANSFER CLEARINGHOUSE
SOFTWARE PRODUCTIVITY CONSORTIUM, INC.
**SPC BUILDING**
**2214 ROCK HILL RD**
**HERNDON VA 22070–9858**